CPEN 311: Digital Systems Design

Verilog Loops, Generates, Tri-States

Introduction to Slide Set

1. For loops:

   Under certain conditions you can use for loops in a process / always block. We'll talk about when you can do this, and why you might want to.

2. Generate Statements:

   Allows you to efficiently specify array-type circuits structurally

3. Tri-State Drivers:

   Buses and tri-state drivers are an important part of any digital system. We will consider a tri-state driver, and show how they are implemented in Verilog.

Topic 1: Loop Statements

Behavioural FOR Loop

Verilog contains a behavioural **FOR** loop construct

Verilog

```
for(<initial_assignment>; <expression>; <step_assignment>)
begin
 <sequential statements>
end
```

- The ***for*** loop is a **SEQUENTIAL STATEMENT** and therefore can **ONLY** be used inside of a process/always block.
- **Loops are synthesizable only when**
  1. The loop range is constant at synthesis time
  2. Does not contain WAIT statements

Synthesis of FOR Loops

Synthesizable only when the loop range is constant at synthesis time

Verilog

```
reg [2:0] SW;
integer i;
reg P;

...

always @(SW)
  begin
     P = 0;
     for (i=0;i<3;i=i+1) begin
          P = P ^ SW[i];
     end
   end
```

- Synthesis tool actually **UNROLLS** loop before starting synthesis
- In other words, it tries to replace the loop code with equivalent code before proceeding with synthesis

Loop Unrolling

```
reg [2:0] SW;
integer i;
reg P;

...

always @(SW)
  begin
     P = 0;
     for (i=0;i<3;i=i+1) begin
          P = P ^ SW[i];
     end
   end
```

Loop gets unrolled to (replaced by):

```
P = P ^ SW[0];
P = P ^ SW[1];
P = P ^ SW[2];
```

- For synthesis, it's almost better to think of a loop as being shorthand for replacing some repetitive regular behavioural code like above
- For simulation/testbenches, there are no such restrictions

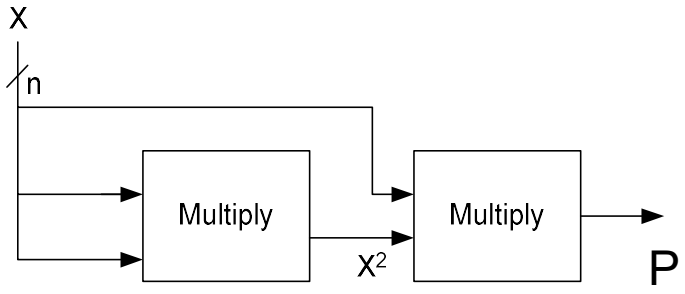
By the way, this example could be replaced by:

```
P = ^SW;   // this means xor all the bits of SW together
```

But the point of the example was to show you how to do a loop.

Build an X^3 Circuit

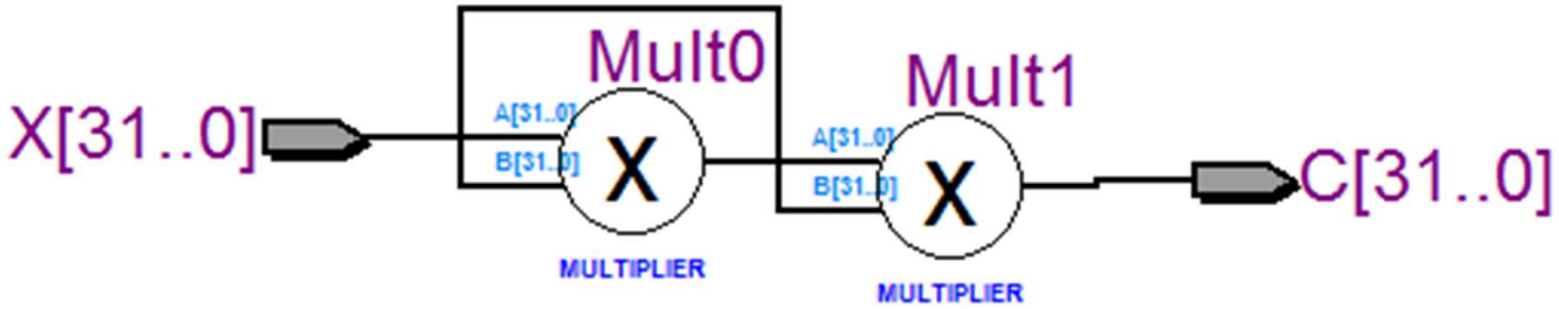
You can describe this as a combinational process using a synthesizable ***for*** loop because the number of iterations is constant




```
integer i;
reg [31:0] P;

...

always @(X)
  begin
     P = 1;
     for (i=0;i<3;i=i+1)
               P = P * X;
  end
```

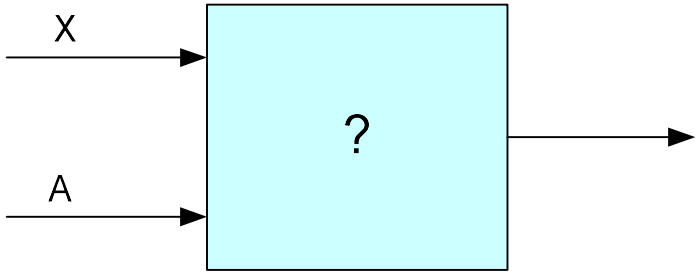
```
integer i;
reg [31:0] P;

...

always @(X)
  begin
     P = 1;
     P = P * X;
     P = P * X;
     P = P * X;
   end
```

From Tools->Netlist Viewer->RTL Viewer

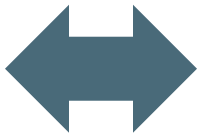


Recall the X^A Circuit

You **CANNOT** describe this using for loop because number iterations is not constant




```
integer i, A;
reg [31:0] P;

...

always @(X)
  begin
     P = 1;
     for (i=0;i<A;i=i+1)
               P = P * X;
  end
```



```
integer i;
reg [31:0] P;

...

always @(X)
  begin
     P = 1;
     ???
   end
```

Looping Behaviour

Observation

Since the bounds must be known at compile time, a loop in a process is really just a short-cut to describe behaviour over a fixed number of iterations. It does not really imply any looping behaviour of the hardware.

True Looping Behaviour

True Looping behaviour (variable iterations) of the hardware requires a datapath.

For Loops In General

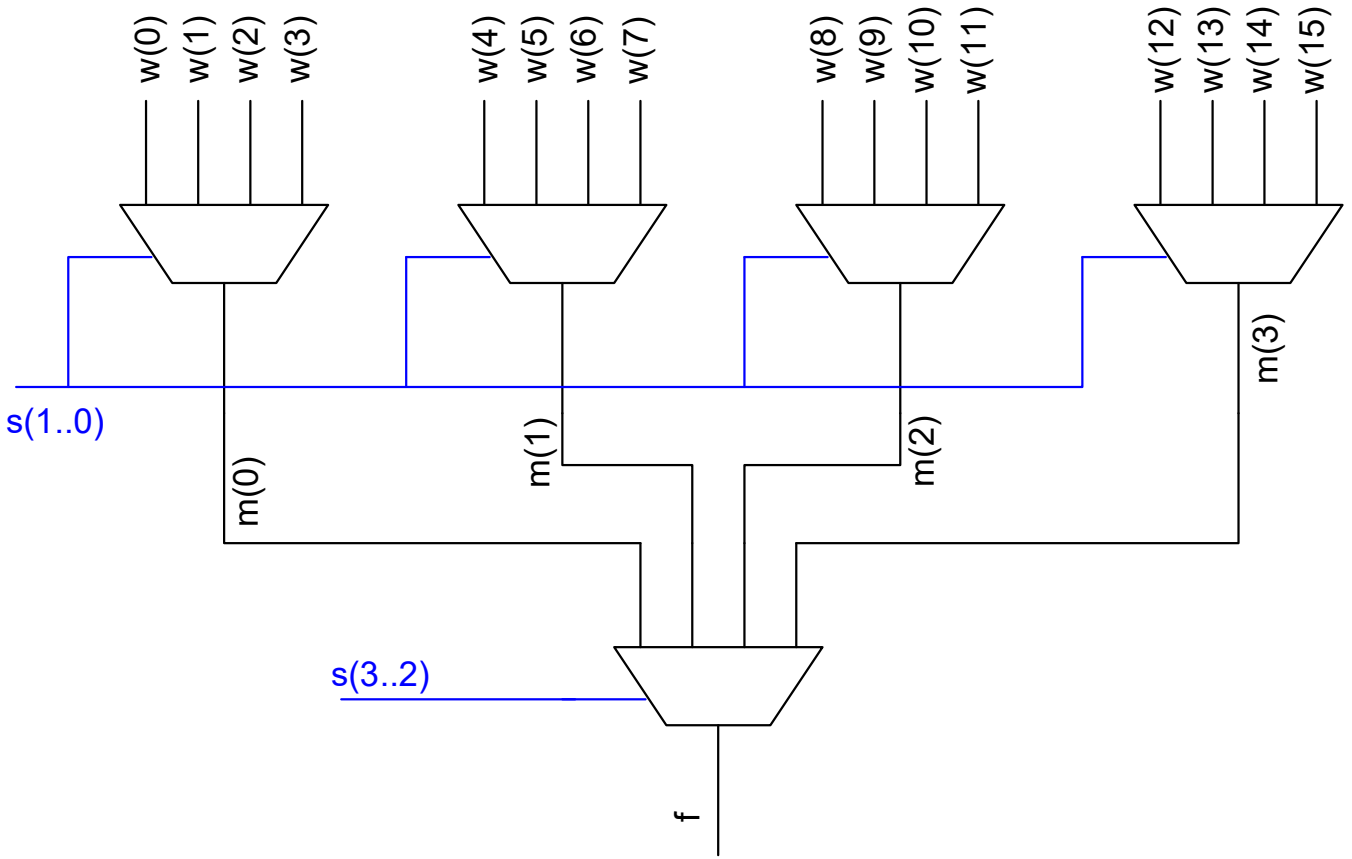
Ok to use in simulation and testbenches (they are quite handy!)

Ok in synthesizable code if bounds are fixed.

Topic 2: Generate Statements

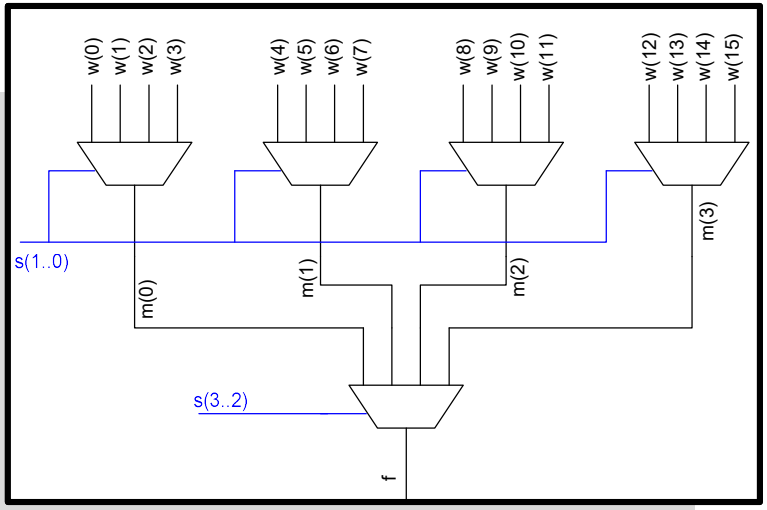
Structural Descriptions

Suppose we want to design this circuit:



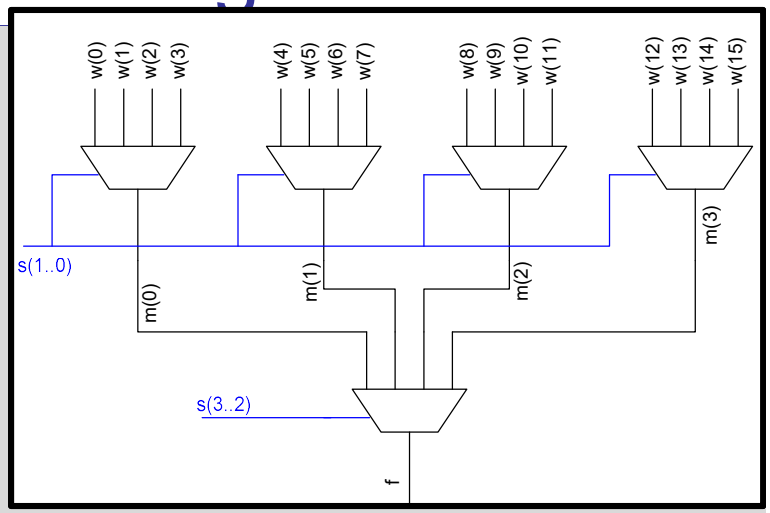
Further, suppose we already have a 4-input 1-bit mux component and we want to create a structural description

A Structural Description: Verilog




```
module mux_16to1 (W, S, F);
input [15:0] W;
input [3:0] S;
output F;

wire [3:0] M;

// Second Stage Mux
mux_4to1 mux4 (M[3], M[2], M[1], M[0], S[3:2], F);

// First Stage Mux
mux_4to1 mux3 (W[12], W[13], W[14], W[15], S[1:0], M[3]);
mux_4to1 mux2 (W[8], W[9], W[10], W[11], S[1:0], M[2]);
mux_4to1 mux1 (W[4], W[5], W[6], W[7], S[1:0], M[1]);
mux_4to1 mux0 (W[0], W[1], W[2], W[3], S[1:0], M[0]);

endmodule
```

Using Generate: Verilog



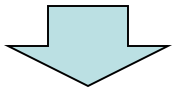

```
module mux_16to1 (W, S, F);
input [15:0] W;
input [3:0] S;
output F;

wire [3:0] M;

// Second stage Mux
mux_4to1 mux4 (M[3], M[2], M[1], M[0], S[3:2], F);

// First stage muxes
genvar i;

generate
   for (i=0; i < 4; i=i+1) begin: big_mux
          mux_4to1 mux ( W[4*i], W[4*i+1],
                         W[4*i+2], W[4*i+3], S[1:0], M[i] );
   end
endgenerate

endmodule
```

GENERATE Statement

- Shorthand to instantiate components with a regular (uniform) connectivity pattern
- Generates are **CONCURRENT STATEMENTS** and can only be used in an architecture (cannot be in a process).
- Looks similar to the behavioural FOR loop from, **but quite different**
- Synthesis tool **UNROLLS** any Generate For constructs before proceeding with actual synthesis
- Name for Generate is mandatory

Unrolling:

```
genvar i;

generate
   for (i=0; i < 4; i=i+1) begin: big_mux
           mux_4to1 mux ( W[4*i], W[4*i+1],
                          W[4*i+2], W[4*i+3], S[1:0], M[i] );
   end
endgenerate
```

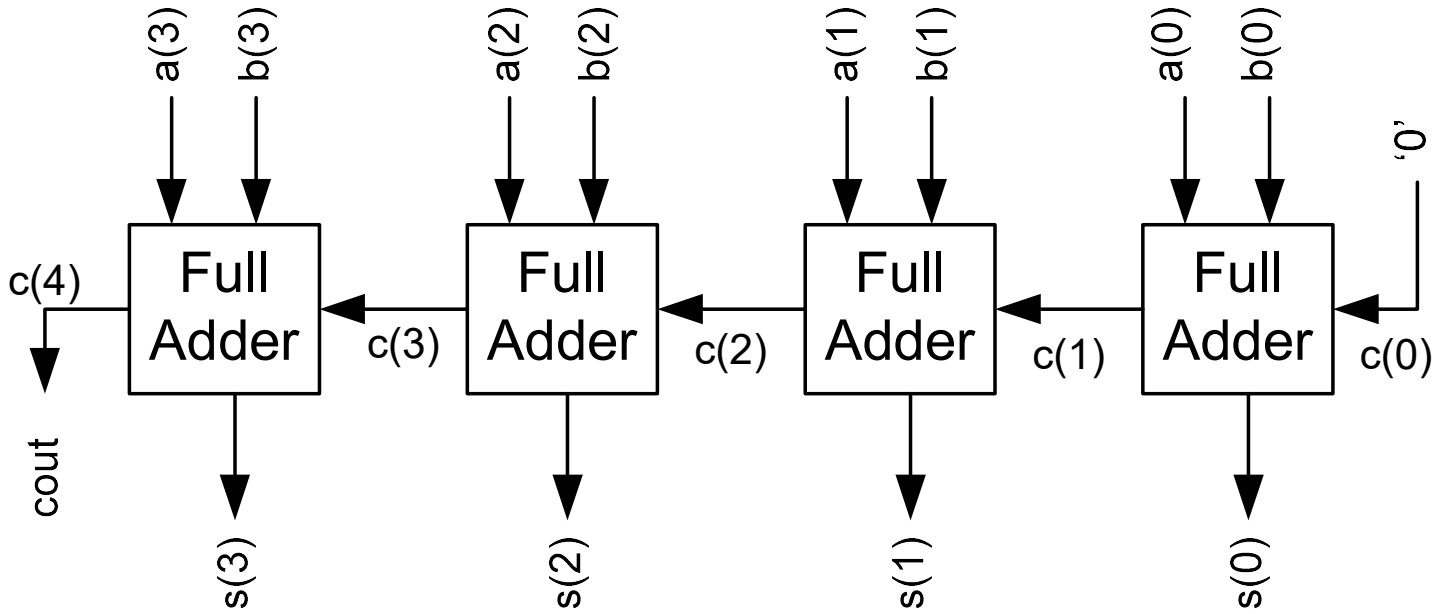


```
mux_4to1 mux3 (W[12], W[13], W[14], W[15], S[1:0], M[3]);
mux_4to1 mux2 (W[8], W[9], W[10], W[11], S[1:0], M[2]);
mux_4to1 mux1 (W[4], W[5], W[6], W[7], S[1:0], M[1]);
mux_4to1 mux0 (W[0], W[1], W[2], W[3], S[1:0], M[0]);
```

Synthesis tool **replaces** the **Generate** statement with the unrolled version before proceeding to actual synthesis

Another Example: 4-bit Adder

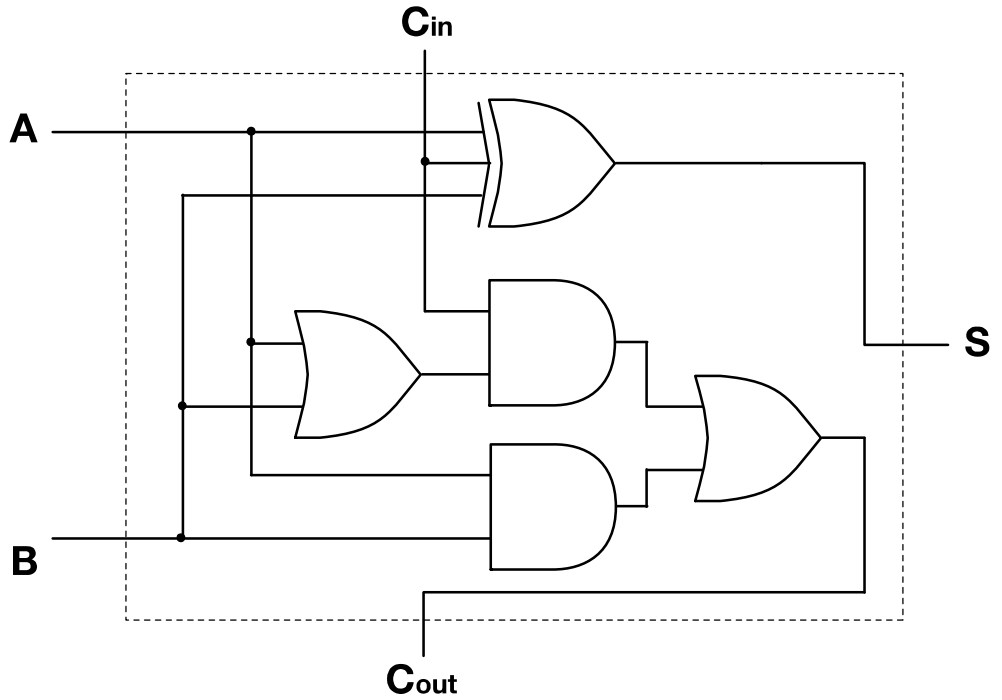
Consider a 4-bit adder constructed from components of Full Adders



Full Adder takes three 1-bit inputs (A, B, Cin) and adds them to produce two 1-bit outputs (Cout, S).

Full Adder

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

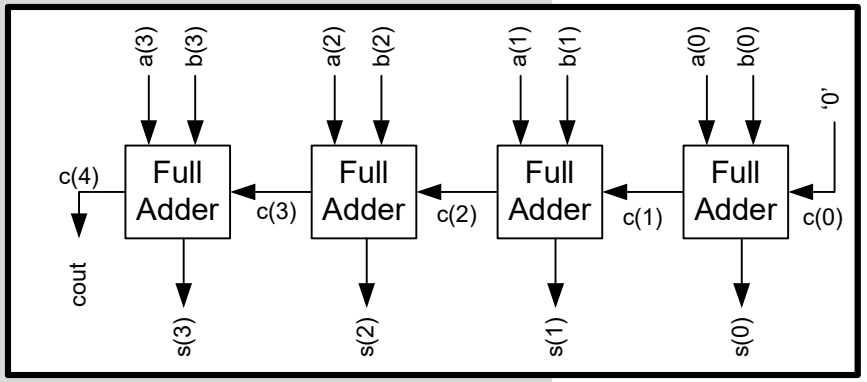


Verilog

```
module full_adder (A, B, Cin,
                   S, Cout);
input A, B, Cin;
output S, Cout;

assign S = A ^ B ^ Cin;
assign Cout = (A & B) |
        (B & Cin) | (A & Cin);
endmodule;
```

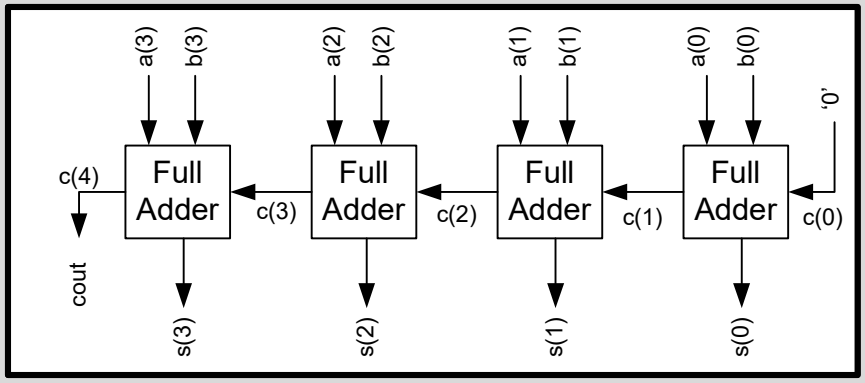
A Structural Description: Verilog

```
module add4 (A, B, S, COUT);
input [3:0] A, B;
output [3:0] S;
output COUT;

wire [4:0] C;

assign COUT = C[4];
full_adder FA0 (A[0], B[0], C[0], S[0], C[1]);
full_adder FA1 (A[1], B[1], C[1], S[1], C[2]);
full_adder FA2 (A[2], B[2], C[2], S[2], C[3]);
full_adder FA3 (A[3], B[3], C[3], S[3], C[4]);
assign C[0] = 1'b0;

endmodule
```



Using ***Generate*** (Verilog)

```
module add4 (A, B, S, COUT);
input [3:0] A, B;
output [3:0] S;
output COUT;

wire [4:0] C;
genvar i;

assign COUT = C[4];
generate
   for (i=0; i < 4; i=i+1) begin: fa
           full_adder FA ( A[i], B[i], C[i], S[i], C[i+1]);
   end
endgenerate
assign C[0] = 1'b0;

endmodule
```

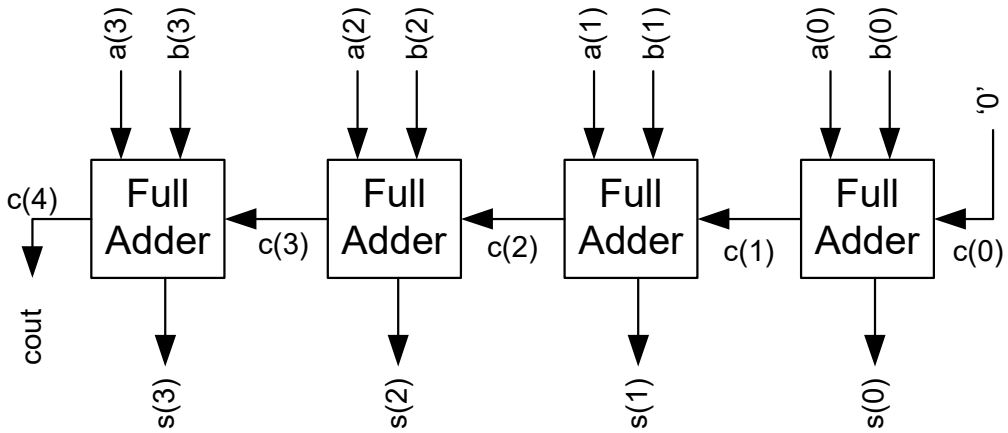


Ifs inside ***GENERATEs***

Often times, when there is an opportunity to use a ***Generate***, there are irregularities in a few of the instances

For example, in the Adder example,

- the 0-th instance has an irregular carry-in (it is connected to '0')
- the last instance has an irregular carry-out (it connects to a port)



We dealt with this using some signal assignments. But we could also handle these "special cases" using a **Generate If** statements

```
assign COUT = C[4];
generate
   for (i=0; i < 4; i=i+1) begin: fa
           full_adder FA ( A[i], B[i], C[i], S[i], C[i+1]);
   end
end generate
assign C[0] = 1'b0;
```

```
genvar i;
generate
   for (i=0; i < 4; i=i+1) begin: fa
       if (i == 0)
           full_adder FA ( A[i], B[i], 1'b0, S[i], C[i+1]);
       else if (i==3)
           full_adder FA ( A[i], B[i], C[i], S[i], COUT);
       else
           full_adder FA ( A[i], B[i], C[i], S[i], C[i+1]);
   end
endgenerate
```

Too many “special cases” defeats the elegance of Generates

Use with discretion

Important Reminder

Generate For and the Behavioural **For** appear similar but are different

Generate If and the Behavioural **If** appear similar but are different

Generate For and **Generate If** are **CONCURRENT STATEMENTS**

- They can **only** be used **in a module** and **outside of a process**
- Used as a convenience for describing a large number of component instantiations with a regular pattern

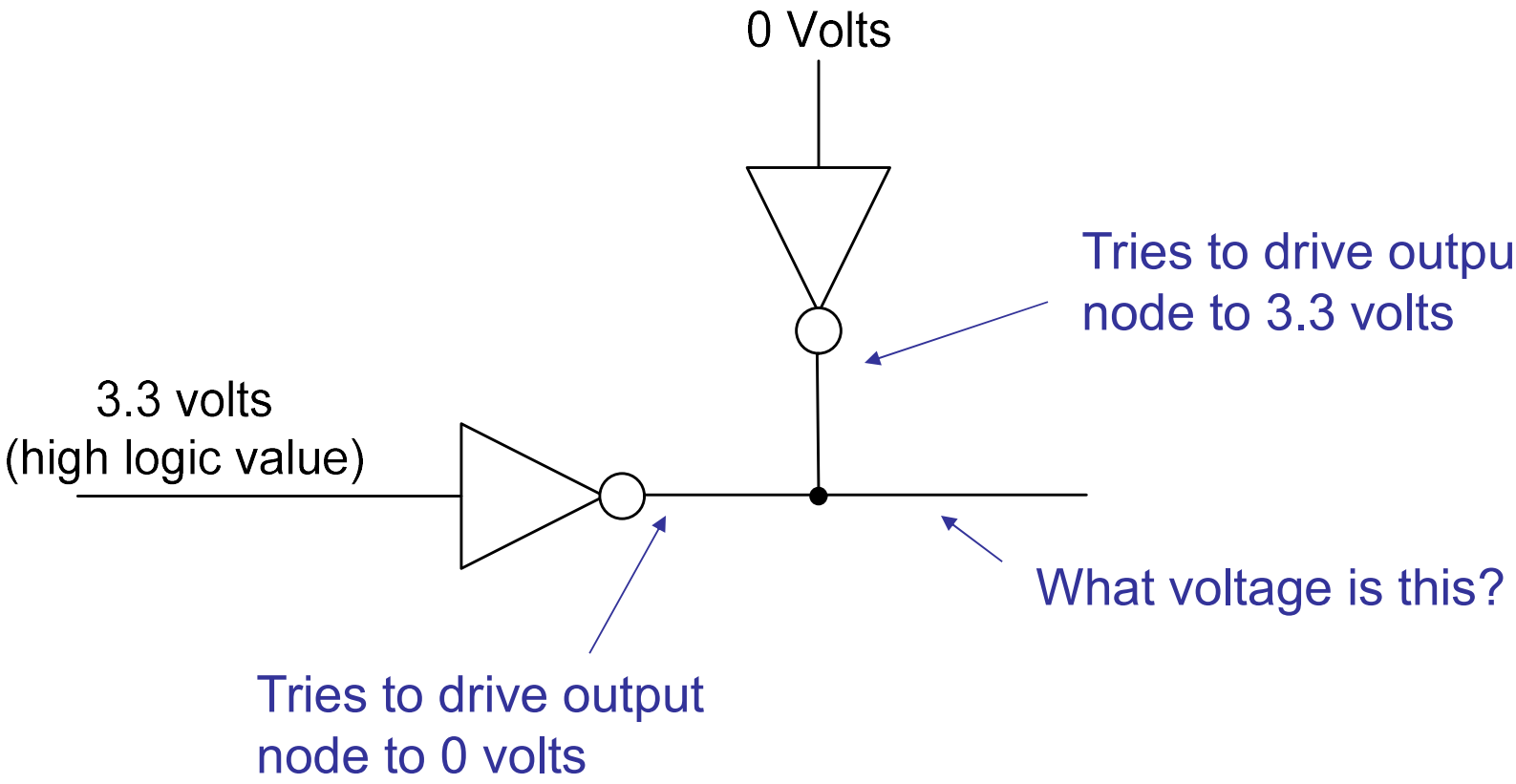
Behavioural **For** and Behavioural **If** are **SEQUENTIAL STATEMENTS**

- They can **only** be used **in an always process** and **not outside**
- Used as a convenience for describing a series of similar behaviour

Neither ***For***'s imply any looping in the hardware. We need specialized datapaths and controllers for this.

Topic 3: Tri-State Logic

Multiple Drivers



- What actually happens depends on the transistor level implementation
- For CMOS design style (almost all circuits are CMOS), this leads to high current that would damage the chip

Simulation

Now think of what happens if we simulate a Verilog specification of this circuit…

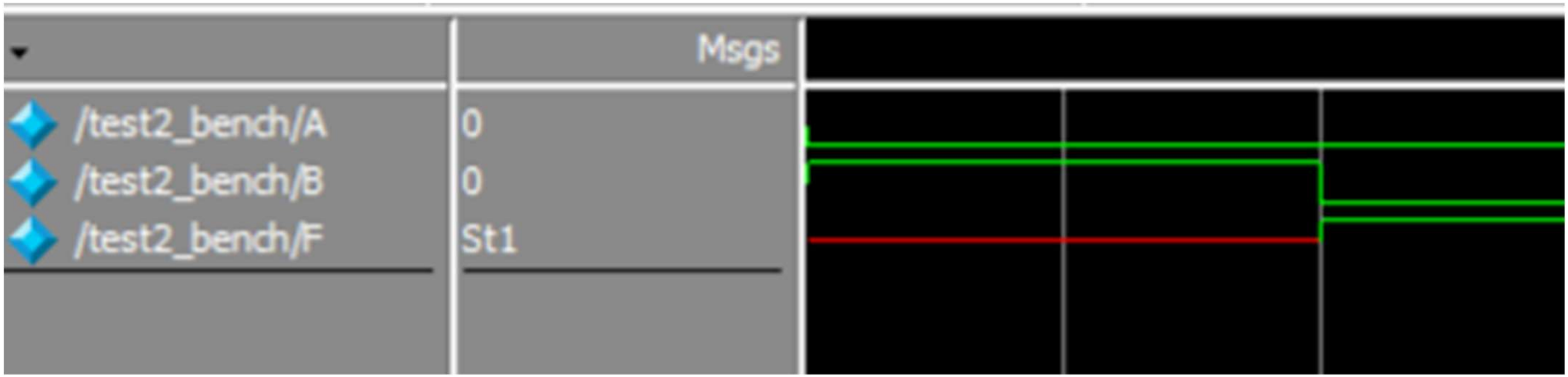
What should the value of this output signal be?

- One possible value is ‘x’
- ‘x’ stands for unknown

When the Verilog simulator sees ‘0’ and ‘1’ driven onto the same signal, it sets the value of the signals’ value to ‘x’.
That tells you something is wrong.

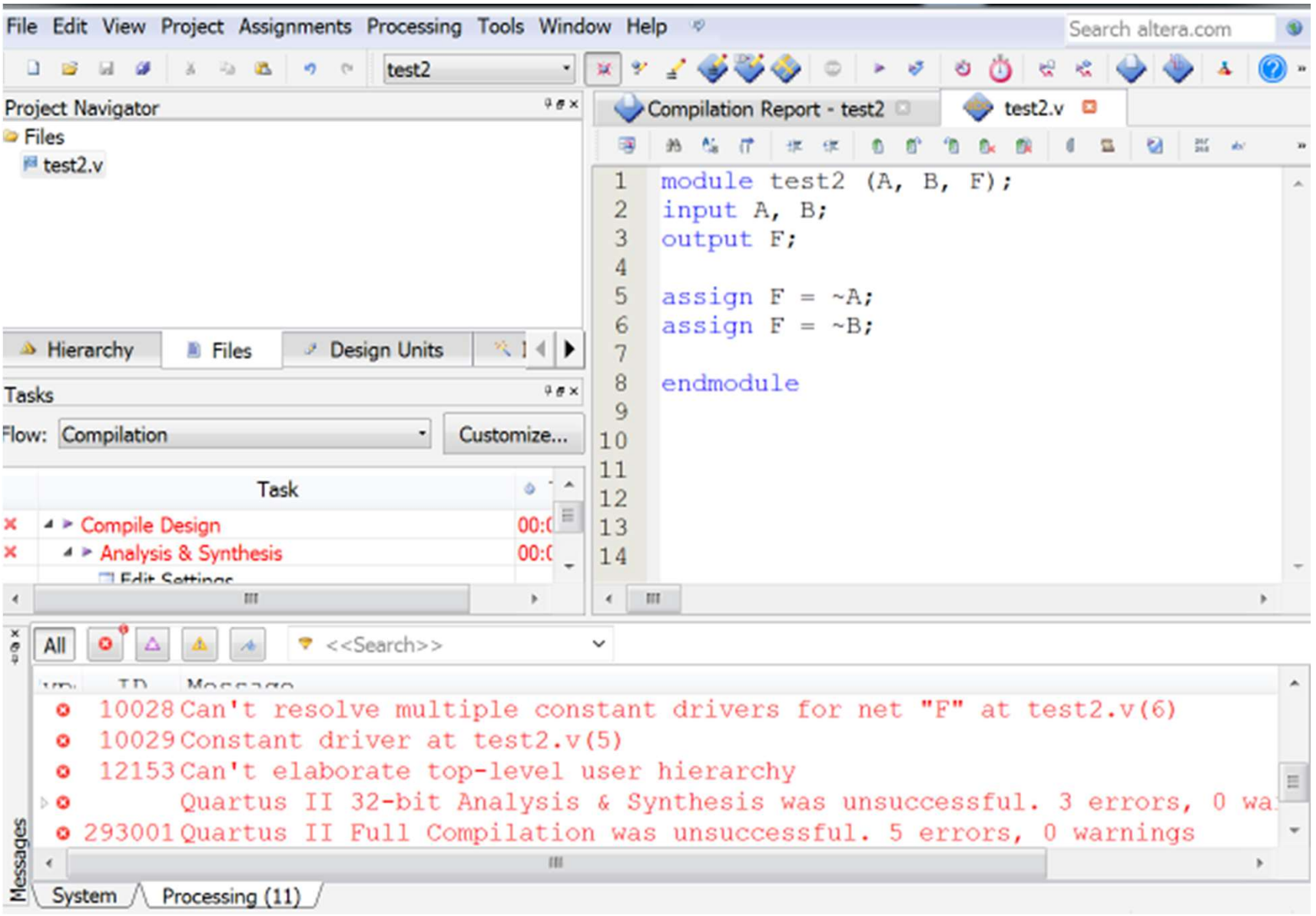
Remember, in the real circuit, the node will have some voltage…
(can’t measure an ‘x’ in the lab)

ModelSim

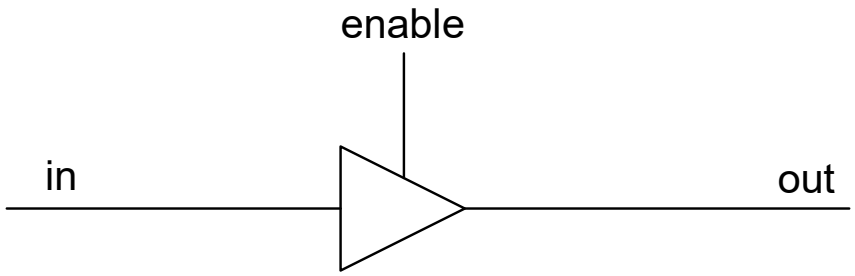



```
module test2 (A, B,
F);
input A, B;
output F;

assign F = ~A;
assign F = ~B;

endmodule
```

If you try to synthesize this in Quartus: It tells you there is an error and doesn't even let you implement it:



Tri-State Drivers



If **ENABLE** is 1, then **OUT** is driven with the value on in.

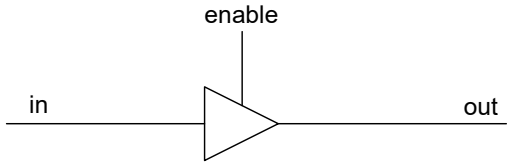
If **ENABLE** is 0, then **OUT** is **NOT DRIVEN**.

What does that mean?

If no one else is driving this signal, then the signal is **FLOATING**

Simulation

Now think of what happens if we simulate a Verilog specification of this circuit…



If ENABLE is zero, what should the simulation say the output is?

- One possible value of a **REG/WIRE** is ‘z’
- ‘z’ stands for high impedance

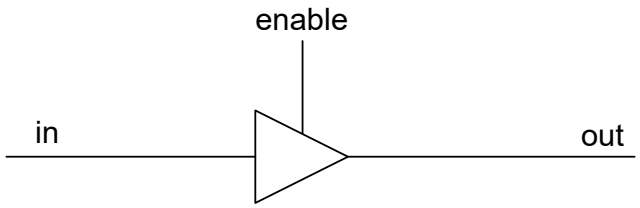
in	enable	out
0	0	z
1	0	z
0	1	0
1	1	1

Remember: in the real world, a signal will have a voltage, even if it is not being driven

(Can’t measure z in the lab)

Describing Tristate Driver in Verilog

Would this work?




```
module tristate_driver (I,
                        ENABLE, F);
input I, ENABLE;
output reg F;

always @(*)
   if (ENABLE == 1)
      F <= I;

endmodule
```

No! This describes a latch!

Recall: The Evil Inferred Latches

```
module tristate_driver (I,
                        ENABLE, O);
input I, ENABLE;
output reg O;

always @(*)
   if (ENABLE == 1)
      O <= I;

endmodule
```

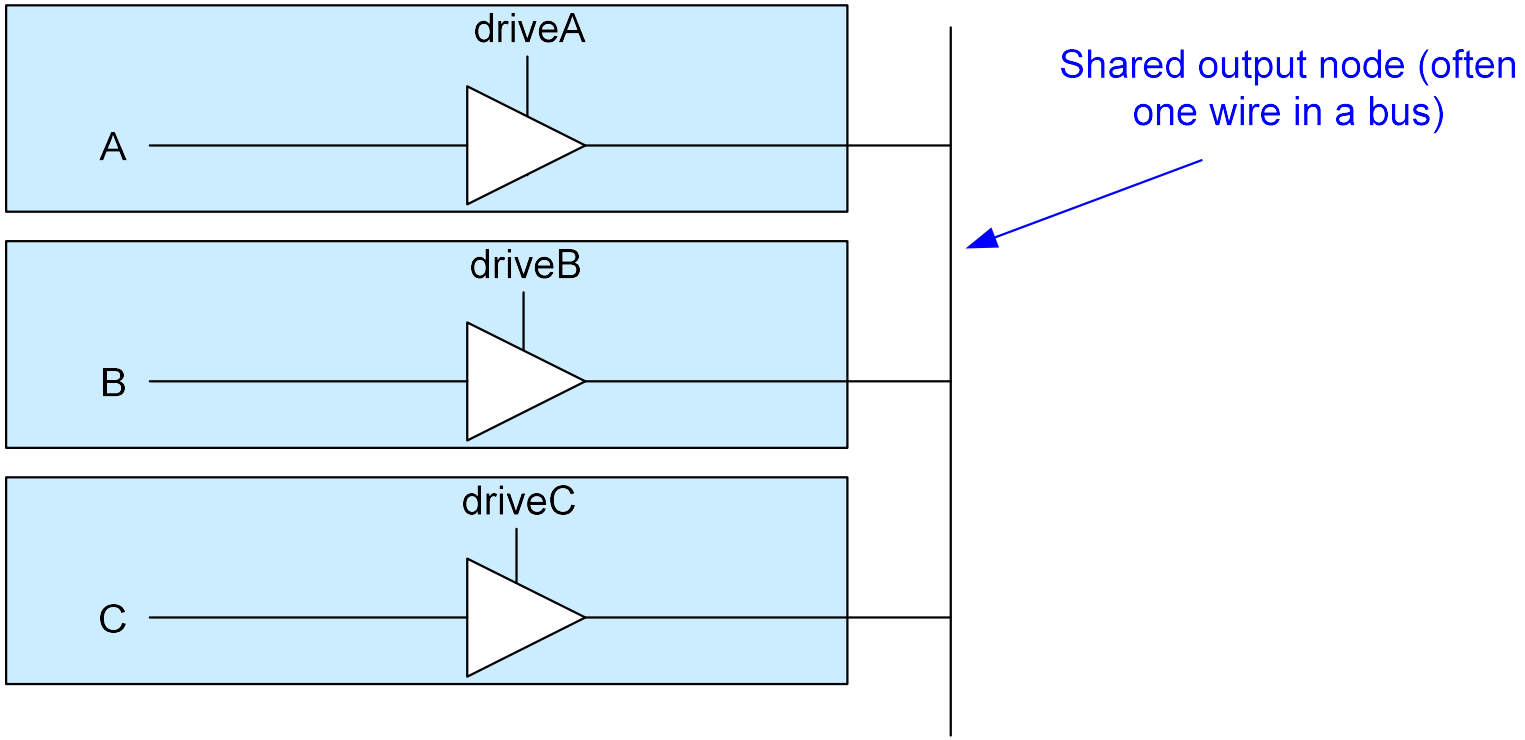
- If **ENABLE** is zero, **O** is not assigned a value.
- The Verilog LRM says that in this case, the output maintains its old value. → Inferred Latch

The Correct Way

```
module tristate_driver (I,
                        ENABLE, O);
input I, ENABLE;
output reg O;

always @(*)
   if (ENABLE == 1)
      O <= I;
   else
      O <= 1'bz;

endmodule
```

Synthesis tools actually understand that a physical wire can take on values of **High Impedance**

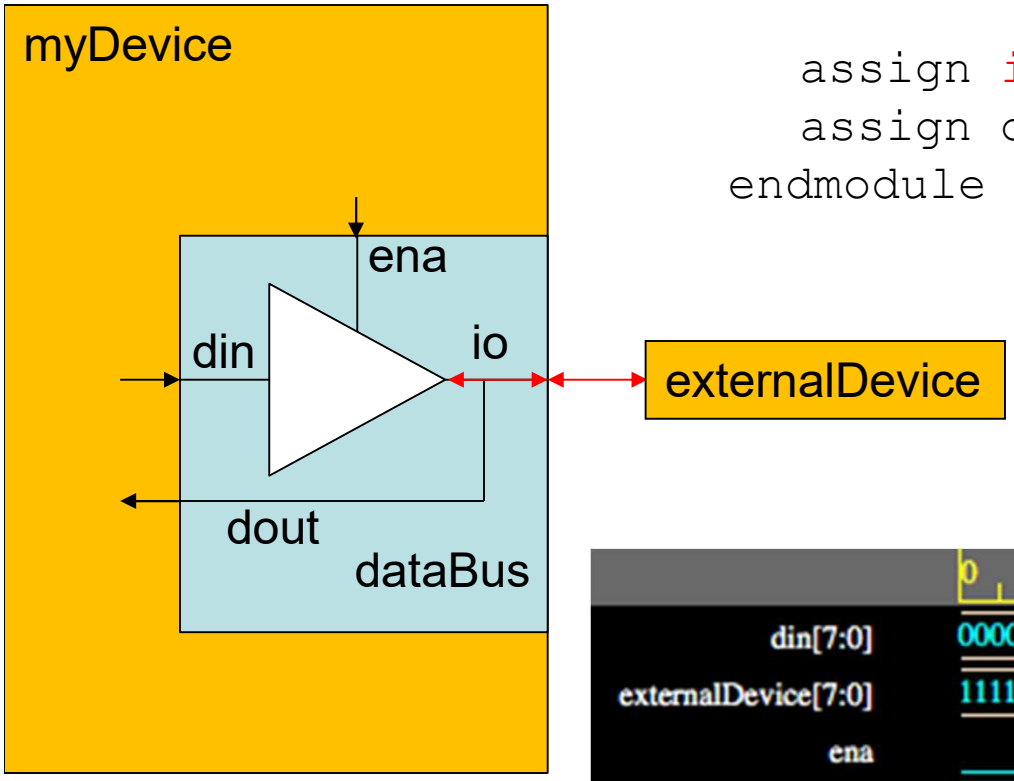
How a tri-state driver is used in digital circuits



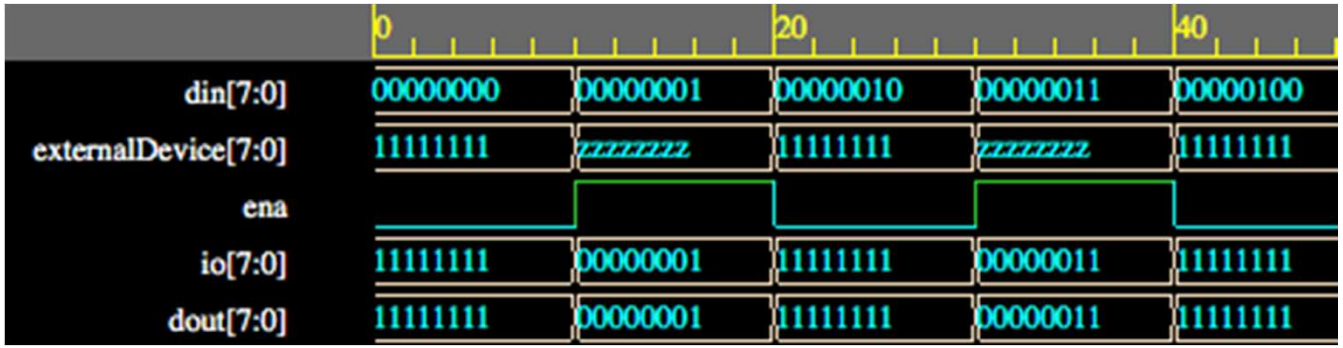
- At most one of the tri-state drivers turns on at a time
- If we want A to drive the output, turn on driveA, etc.
- Similar behaviour to a Multiplexer
- So actually, the only time when multiple drivers is OK is when you are using tristate drivers

Bidirectional pins (eg, data bus)

```
module dataBus (ena, din, dout, io);
   input  logic ena;
   input  logic [7:0] din;
   output logic [7:0] dout;
   inout  wire  [7:0] io;

   assign io = ena ? din : 8'bZ;
   assign dout = io;
endmodule
```



Note: externalDevice also needs to know **when** to be in tri-state mode **so it does not drive data at the same time (a "bus fight")**



Final Comments on Tristate

- At current transistor technology nodes, people typically do not use tristate logic in their designs (speed issues)
- Tristate is still used for off-chip I/O
- Modern FPGAs have no ability to create internal tristate drivers
  - Can only be used for off-chip I/O signals
  - On-chip, Quartus would likely convert your tristate logic to multiplexer networks

Summary of Slide Set

1. For loops:

   Under certain conditions you can use for loops in a process/always block. We talked about when you can do this, and why you might want to.

2. Generate Statements:

   Allows you to efficiently specify array-type circuits structurally

3. Tri-State Drivers:

   Buses and tri-state drivers are an important part of any digital system. We considered a tri-state driver, and showed how they are implemented in VHDL and Verilog